Documentation for random.h and random.c

Steven Andrews, © 2003
See the document "LibDoc" for general information about this and other libraries.

```
#ifndef __random_h
#define __random_h

#include <time.h>
#include <stdlib.h>
#include <math.h>

#define RAND_MAX_30 1073741823
#if RAND_MAX==32767
   #define rand30() ((long)rand()<<15|rand())
#elif RAND_MAX<RAND_MAX_30
   #define rand30() ((rand()&32767L)<<15|rand()&32767L)
#else
   #define rand30() (rand()&RAND_MAX_30)
#endif

#define randomize() srand((unsigned int) time(NULL))
#define exprand(a) (-log((rand()+1.0)/(RAND_MAX+1.0))*(a))
#define exprand30(a) (-log((rand30()+1.0)/(RAND_MAX_30+1.0))*(a))
#define unirand(lo,hi) ((float)rand()/RAND_MAX*((hi)-(lo))+(lo))
#define unirand30(lo,hi) ((double)rand30()/RAND_MAX_30*((hi)-(lo))+(lo))
#define signrand() (rand()&1?1:-1)
#define coinrand(p) (rand()<(RAND_MAX+1.0)*(p))
#define coinrand30(p) (rand30()<(RAND_MAX_30+1.0)*(p))
#define intrand(n) (rand()%(n))
#define intrand30(n) (rand30()%(n))
#define thetarand() (acos(1.0-2.0*rand()/RAND_MAX))
float binomrand(int n,float m,float s);
int intrandp(int n,float *p);
int poisrand(float xm);
float gaussrand();
void randtable(float *a,int n,int eq);
void showdist(int n,float low,float high,int bin);

#endif
```

Include: `<stdio.h>, <time.h>, <math.h>`
Example program: `LibTest.c`

History: Written 5/12/95; modified 11/12/98. Routines have been moderately tested. Works with Metrowerks C. Documentation updated 10/16/01. Ported to Linux 10/16/01. Added `randtable` 11/16/01. Added `intrandp` 1/14/02. Added `poisrand` 1/26/02. Added `rand30` and other `…30` functions 11/8/02, but didn't document them until 9/2/03. Added `gaussrand` 4/24/03. Added `intrand30`, `coinrand30`, `RAND_MAX_30`, and improved `unirand` and `unirand30` 9/2/03.

Most of these routines return random numbers, chosen from a variety of densities. They use the stdlib.h `rand()` function, and have not been analyzed for the randomness quality. Before using these routines, it is recommended that the random number generator seed be set with either the stdlib.h function `srand(unsigned int seed)` or set to the clock value with `randomize`.

Note that `1.0*rand()/RAND_MAX` returns a uniform density on [0,1] and `(rand()+1.0)/(RAND_MAX+1.0)` is uniform on (0,1]. To convert these uniform densities to the density $\rho(x)$, first calculate the cumulative probability $P(x)=\int_{-\infty}^{x}\rho(x')dx'$, where it is seen that $P(x)$ is 0 at x=–∞ and 1 at x=∞. Now if the value for $y=P(x)$ is chosen with a uniform density, its value mapped onto x has the desired density. Thus a function should return $x=P^{-1}(y)$. It is also helpful to know that the CodeWarrior compiler on a Macintosh has `RAND_MAX` equal to $2^{15}-1$, whereas it is $2^{31}-1$ for the gcc compiler on Linux. The routines that end with a "`30`" are especially helpful on a Macintosh (and hinder slightly on Linux) by allowing 2^{30} possible random numbers.

The table below shows the domain, range, and densities of the macros and routines given here. The domains are the domains over which the functions give reasonable values, but are not neccessarily sensible. For example, `coinrand` can accept an input anywhere between –∞ and ∞, although the function always returns 0 if `p`<0 and 1 if `p`>1. The densities are only strictly correct in the limit that `RAND_MAX` approaches infinity. In regions where the density is small (where $\rho(x)\Delta x\approx 1/$`RAND_MAX`, for some characteristic Δx), a small set of random numbers is mapped to a large output range, leading to relatively sparse coverage.

Name	Domain	Range	Density
`exprand`	[0,∞)	[0,∞)	$1/a*\exp(-x/a)$
	(–∞,0]	(–∞,0]	$1/a*\exp(-x/a)$
`exprand30`	[0,∞)	[0,∞)	$1/a*\exp(-x/a)$
	(–∞,0]	(–∞,0]	$1/a*\exp(-x/a)$
`unirand`	(–∞,∞);`lo`≠`hi`	[`lo`,`hi`]	1/|`hi`–`lo`|
`unirand30`	(–∞,∞);`lo`≠`hi`	[`lo`,`hi`]	1/|`hi`–`lo`|
`signrand`		{–1,1}	{0.5,0.5}
`coinrand`	(–∞,∞)	{0,1}	{1–`p`,`p`}
`coinrand30`	(–∞,∞)	{0,1}	{1–`p`,`p`}
`intrand`	[1,∞)	{0,1,…,`n`–1}	{1/`n`,1/`n`,…,1/`n`}
`intrand30`	[1,∞)	{0,1,…,`n`–1}	{1/`n`,1/`n`,…,1/`n`}
`thetarand`		[0,π]	1/2*sin(x)
`binomrand`	`n`>0, all `m`,`s`	[`m`–`s`√(3`n`),`m`+`s`√(3`n`)]	≈Gaussian with mean `m`, std. dev. `s`
`intrandp`	`n`>0, 0≤p_i≤1	{0,1,…,`n`–1}	{p_0,p_1-p_0,…,1-p_{n-2}}
`poisrand`	(–∞,∞)	[0,∞)	Poisson with mean `xm`
`gaussrand`		(–∞,∞)	Gaussian with mean 0, std. dev. 1

`randomize` sets the random number generator seed to the current time.
`exprand` returns an exponentially distributed random double.
`exprand30` is identical to exprand, except it uses a 30 bit random number.
`unirand` returns a uniformly distributed double between `lo` and `hi`, inclusive.
`unirand30` is identical to `unirand`, except it uses a 30 bit random number.
`signrand` returns 1 or –1 with equal probability.
`coinrand` returns 1 with probability p, and 0 otherwise.
`coinrand30` is identical to `coinrand`, except is uses a 30 bit random number.
`intrand` returns an integer between 0 and `n-1` with equal probability for each value. The probability distribution is correct if `n` is a divisor of `RAND_MAX+1` (*i.e.* an integer power of 2), quite good if `n` is a small integer, and poor if `n` is a significant fraction of `RAND_MAX` and not a divisor of `RAND_MAX+1`.
`intrandp` is identical to `intrand`, except it uses a 30 bit random number.

`thetarand` is intended for use in choosing a random θ direction in spherical coordinates.

`binomrand` adds together `n` random variables from a uniform density and then scales the sum to yield the proper mean and standard deviation. It's an easy alternative for a true Gaussian density, although not as fast or as well distributed as a look-up table and interpolation (see `randtable`). It's also misnamed, since a true binomial distribution is the sum of numbers chosen from {0,1}.

`intrandp` is similar to `intrand`, but allows non-uniform probabilities for each integer (however, it doesn't improve on the distribution accuracy for large `n` values). `p` is sent in as a list of cumulative probabilities for each integer. Since they are cumulative, `p` is an increasing list of numbers between 0 and 1, and p_{n-1} should be equal to 1. Results will always be between 0 and `n-1`, even with incorrect `p` values.

`poisrand` returns an integer chosen from a Poisson density with mean `xm`, which will typically be in the range $xm \pm \sqrt{xm}$. This routine is copied almost verbatim from *Numerical Recipies*. A feature which the book routine has and which is kept here is that if the routine is called more than once with the same value of `xm`, it doesn't recalculate some variables, in order to speed up the routine. Negative values of `xm` are possible but always return a value of 0.

`gaussrand` returns a normal deviate with mean 0 and standard deviation 1 using the Box-Muller transformation described in *Numerical Recipies*.

`randtable` fills in a lookup table with entries for quickly converting a uniform density to an alternate density, using `eq` to indicate which density is desired. `n` is the number of elements in the table. If `eq` is 1, the density is a normal density with mean 0 and standard deviation 1; returned values range from $-\text{erf}^{-1}(0.5/n-1)$ to $\text{erf}^{-1}(0.5/n-1)$. For example, if `rt` is a table with 1024 elements, the following expression would return a normally distributed random variable with mean `mu` and standard deviation `sd`: `x=mu+sd*rt[rand()&1023]`, also the range is from –2.33 to 2.33. Clearly, there are only `n` possible outcomes in this expression, which could be corrected by linear interpolation and somewhat slower and lengthier code.

`showdist` is only intended for debugging routines such as `binomrand`, so it is not a general routine. It plots a bar graph (`bin` bars that range from the first bar center at `low` to the last bar center at `high`) showing the distribution of `n` random variables from `binomrand(10,0,1)` or some other routine; it also displays the actual mean and standard deviation.